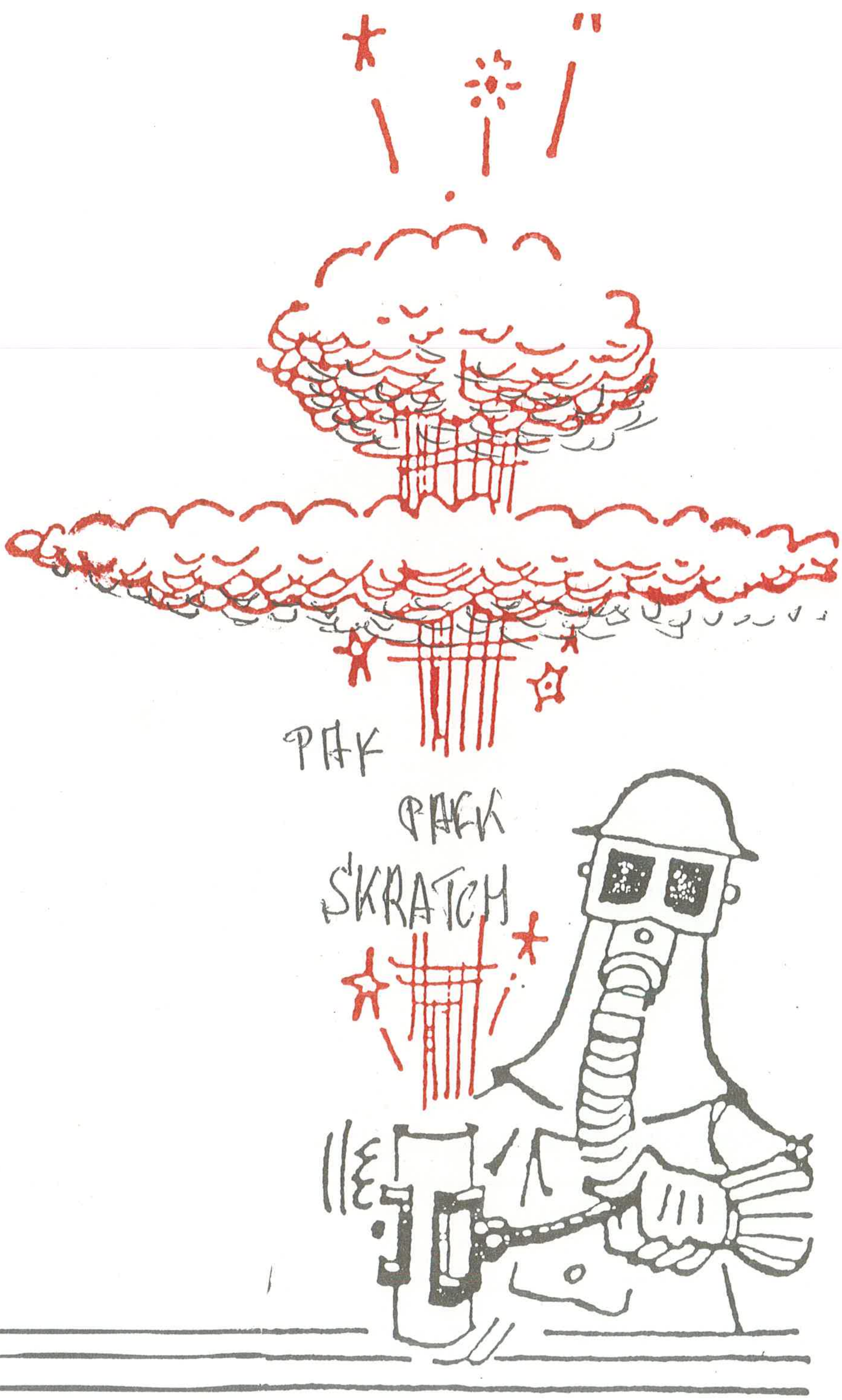
JEDI

49

LE JOURNAL QUI VOUS MITONNE DE BONNES RECETTES

JANVIER 1989



COUP DE GUEULE

Pour une fois, mon éditorial s'appelle "coup de gueule". En novembre et décembre, la grève des postiers immobilisait le courrier, et cette situation a commencé à se débloquer vers le 15 décembre. Et que ne reçoit-je pas parmi diverses missives, prospectus et abonnements, (TELERAMA des deux semaines précédentes, merci pour être informé des programmes intéressant que j'aurai raté...): une relance EdF, reçue le 21 décembre m'informant que si le montant de ma facture n'était pas réglé avant le 20 décembre, on me couperait le courant... Et d'abord quelle facture? Pourtant, tous les organismes d'état avaient fait savoir par la radio et la télévision que les relances seraient suspendues jusqu'à la régularisation de la distribution du courrier. Ca a peut-être été aussi votre cas. Après information prise par voie téléphonique auprès d'EdF, on m'avait simplement répondu que l'ordinateur avait relancé automatiquement les factures impayées. Comme d'habitude, quand ça cafouille, c'est l'ordinateur! Et leur(s) programmeur(s) si grassement payé(s) chez EdF, ne pouvai(en)t-il(s) pas rajouter une ligne dans le programme, du style:

```
SI grève, ALORS relance=relance+8
```

et recompiler tout le super logiciel de relance pour éviter cette situation. Après réflexion, combien coûte cet oubli (volontaire ou involontaire); supposons qu'un million d'abonnés EdF reçoit la méme relance, ceci pour la seule région parisienne, à 2,00 Fr par lettre, ça fait dans les... 2 Millions de francs!!! Et je suis peut-être en dessous de la réalité. Bigre, voici une faute informatique coûteuse selon mon appréciation (qui n'est peut-être pas celle d'EdF). Y-a-t il beaucoup de patrons de PME qui laisseraient passer une bévue de 2 MF? Par parenthèse, LECLERC sanctionne pour le "vol" de deux fèves; Valjean n'est pas loin, qui a été condamné au bagne pour deux pommes.

Tiens, j'ai envie d'être méchant: le prochain quémandeur quêtant pour les malades du corgnolon, les bébés phoques, les chômeurs de courte durée, les tranfuges du PC, les drogués à l'eau de cologne, les drogués du Minitel, les drogués de "Sacrée Soirée", les marchands de tapis, les perdants au Tapis Vert, les fans de Bernard Tapie soutenant sa candidature et démarcheurs de tout poil qui sonneront à ma porte seront renvoyés chez EdF qui a apparemment les moyens d'être généreux.

SOMMAIRE

Nos coordonnées: ASSOCIATION JEDI 17, rue de la Lancette 75012 PARIS
tel président: (1) 43.40.96.53
tel secrétaire: (1) 49.85.63.67 (SAM*JEDI, bal SECRETAIRE)
télétel: 3615 SAM*JEDI

FORTH : Initiation

LES SUITES DE SYRACUSE.

par Marc PETREMANN

Systèmes: tous systèmes F83, avec restriction partielle de certaines définitions en code machine 8086 (PC ou compatibles)

Note: suite à un courrier déjà ancien de plusieurs mois, j'ai commencé à élaborer une série d'initiation. Mais les circonstances nous pressant pour l'achèvement du manuel TURBO-Forth, j'ai repris cette série en la remodelant pour faire l'objet du manuel. En conséquence, cet article, bien que sous-titré "Initiation", se veut surtout récréatif. Néanmoins, il illustre l'emploi des principales structures de contrôle, l'optimisation en code machine et le développement de programmes à partir d'une idée simple.

Je compte sur vous pour apporter vos critiques, suggestions et compléments à cet article.

La science des mathématiques a réalisé de grands progrès depuis l'utilisation des ordinateurs grâce auquels de nombreuses tâches ont pu être automatisées. En effet, qui pourrait calculer une image tri-dimensionnelle un peu complexe à la main ou même à l'aide d'une calculatrice de poche? Certes, ce n'est pas irréalisable, mais le nombre de données à traiter est tel que seul un ordinateur reste performant.

Mais aussi puissants que soient certains algorithmes de traitement, ils sont à considérer comme de simples machines virtuelles chargées de triturer les nombres, car incapables de raisonner sur les résultats obtenus et les déductions qui peuvent en être tirées.

Il ne faut pas croire que tout traitement numérique permette l'élaboration de théorèmes. De nombreux problèmes restent encore sans solution, dont la recherche des nombres premiers entre autre.

Un autre problème, élémentaire en apparence, est celui des suites de SYRACUSE. Partant d'un algorithme très simple, on se propose de rechercher tous les nombres calculables à partir de cet algorithme:

- pour tout nombre pair, diviser ce nombre par deux
- pour tout nombre impair, multiplier ce nombre par trois et ajouter 1.

A première vue et considérant qu'un nombre sur deux est impair, il y a une chance sur deux pour que le résultat soit multiplié par trois. A chaque nombre sera donc appliqué deux taux:

- un taux de croissance positif légèrement supérieur à trois fois le nombre initial.
- un taux de croissance négatif, égal à la moitié du nombre initial.

Donc, les suites de nombres devraient être perpétuellement croissantes avec des variations en dent de scie, ceci de n à l'infini; c'est ce que suggère la science des statistiques et que nous allons vérifier sur le champ.

Pour déterminer si un nombre est pair ou impair, il suffit de prendre le reste de la division de ce nombre par deux:

```
n 2 MOD      si reste = 1, n est impair
             si reste = 0, n est pair
```

On peut déjà définir un mot délivrant le prochain mot de la suite de SYRACUSE:

```
: PROCHAIN
 ( n1 --- n2; n2=n1*0.5 si n1 pair; n2=n1*3+1 si n1 impair)
  DUP 2 MOD   IF   3 * 1+  ELSE  2/   THEN ;
```

Essayons tout de suite:

```
6 PROCHAIN .   affiche 3
3 PROCHAIN .   affiche 10
10 PROCHAIN .  affiche 5
5 PROCHAIN .   affiche 16, etc...
```

Et comme nous sommes d'un naturel paresseux, créons une définition qui se chargera de répéter automatiquement l'exécution de PROCHAIN en reprenant à chaque réexécution la nouvelle valeur.

```
: SUITE ( n ---)
 BEGIN
    PROCHAIN DUP .          \ affiche nombre de la suite
    KEY UPC ASCII O = NOT   \ répète si appui sur O
 UNTIL ;
```

6 SUITE affiche la suite:

3 10 5 16 8 4 2 1 4 2 1 4 2 1..

7 SUITE affiche la suite:

22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1 4 2 1

Aie! Les premiers résultats contredisent les statistiques. Dès les deux premiers exemples, la suite tend plutôt à décroître, puis à boucler dès qu'elle atteint la valeur 1.

Première constatation: si on tombe sur une puissance de deux, on est piégé et la décroissance est définitive.

Deuxième constatation: le nombre 1 est la limite minimale de la suite. On peut donc réécrire la définition de SUITE en imposant comme point d'arrêt la valeur 1 aux itérations:

```
: SUITE2 ( n ---)
 BEGIN
    PROCHAIN DUP .          \ affiche nombre de la suite
    DUP 1 =                 \ répète si pas égal à 1
 UNTIL DROP ;
```

On définit également un mot permettant de répéter SUITE2 dans un intervalle numérique:

```
: REPETE-SUITE2 ( déb fin ---)
  1+ SWAP
  DO CR ." SUITE DE " I 5 .R ." : "
  \ affiche texte "SUITE DE n: "
    I SUITE2       \ affiche la suite
  LOOP ;
```

Exécutons REPETE-SUITE2 avec un intervalle de recherche compris entre 2 et 50. Au vu des résultats affichés à l'écran et éventuellement imprimés en tapant:

```
PRINTING ON 2 50 REPETE-SUITE2 CR PRINTING OFF
```

on peut tirer une troisième constatation: il n'est pas nécessaire d'extraire la suite de SYRACUSE d'un nombre apparaissant dans une suite déjà traitée. Exemple: 10 apparait dans la suite de SYRACUSE de 3. On peut donc se passer de traiter la suite de SYRACUSE de 10 car elle ne fera que répéter partiellement la suite de SYRACUSE de 3. Ci-après figurent les suites non redondantes:

```
SUITE DE     2: 1
SUITE DE     3: 10 5 16 8 4 2 1
SUITE DE     6: 3 10 5 16 8 4 2 1
SUITE DE     7: 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

SUITE DE     9: 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16
8 4 2 1
SUITE DE    12: 6 3 10 5 16 8 4 2 1
SUITE DE    15: 46 23 70 35 106 53 160 80 40 20 10 5 16 8
4 2 1
SUITE DE    18: 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5
16 8 4 2 1
SUITE DE    19: 58 29 88 44 22 11 34 17 52 26 13 40 20 10
5 16 8 4 2 1
SUITE DE    21: 64 32 16 8 4 2 1
SUITE DE    24: 12 6 3 10 5 16 8 4 2 1
SUITE DE    25: 76 38 19 58 29 88 44 22 11 34 17 52 26 13
40 20 10 5 16 8 4 2 1
SUITE DE    27: 82 41 124 62 31 94 47 142 71 214 107 322
161 484 242 121 364 182 91 274 137 412 206 103 310 155 466
233 700 350 175 526 263 790 395 1186 593 1780 890 445 1336
668 334 167 502 251 754 377 1132 566 283 850 425 1276 638
319 958 479 1438 719 2158 1079 3238 1619 4858 2429 7288
```

3644 1822 911 2734 1367 4102 2051 6154 3077 9232 4616 2308 1154 577 1732 866 433 1300 650 325 976 488 244 122 61 184 92 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
SUITE DE 30: 15 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
SUITE DE 33: 100 50 25 76 38 19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
SUITE DE 36: 18 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
SUITE DE 37: 112 56 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
SUITE DE 39: 118 59 178 89 268 134 67 202 101 304 152 76 38 19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

SUITE DE 42: 21 64 32 16 8 4 2 1
SUITE DE 43: 130 65 196 98 49 148 74 37 112 56 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
SUITE DE 45: 136 68 34 17 52 26 13 40 20 10 5 16 8 4 2 1

SUITE DE 48: 24 12 6 3 10 5 16 8 4 2 1

Quatrième constatation: toute suite de SYRACUSE d'un nombre n situé entre 2 et 50 converge vers 1 après un nombre variable d'itérations. Cette constatation contredit notre première hypothèse selon laquelle un nombre a plus de chance de croître que de décroître.

Existe-t-il un ou plusieurs nombres pour lesquels la suite progresserait vers l'infini? C'est justement cela le mystère de la suite de SYRACUSE. Il n'existe à ce jour aucune démonstration formelle pour affirmer que tout nombre entier traité par la suite de SYRACUSE aboutit obligatoirement à 1. Ce problème semble avoir été formulé dans les années 1930 par Lothar COLLATZ alors élève à l'université de Hambourg et introduit ensuite à l'université de SYRACUSE (Etats Unis) par Helmut HASSE, un collègue de COLLATZ.

Existe-t-il des corrélations entre le nombre d'itérations et la valeur initiale? Pour connaître le nombre d'itérations d'une suite, modifions SUITE2 pour le déterminer:

```
VARIABLE #ITERATIONS
: SUITE3 ( n ---)
 1 #ITERATIONS !  \ initialisation du nombre d'itérations
 BEGIN
    PROCHAIN      \ calcule nombre de la suite
    DUP 1 = NOT   \ répète si pas égal à 1
 WHILE
    1 #ITERATIONS +!   \ incrémente compteur d'itérations
 REPEAT DROP ;

: REPETE-SUITE3 ( déb fin ---)
 1+ SWAP
 DO CR ." SUITE DE " I 5 .R ." : "
 \ affiche texte "SUITE DE n: "
    I SUITE3     \ recherche du nombre d'itérations
    #ITERATIONS @ . ." itérations"
    \ affiche "n itérations"
 LOOP ;

 2 50 REPETE-SUITE 3   affiche

 SUITE DE     2: 1 itérations
 SUITE DE     3: 7 itérations
 SUITE DE     4: 2 itérations
 SUITE DE     5: 5 itérations
...etc...
 SUITE DE    49: 24 itérations
 SUITE DE    50: 24 itérations
```

Le nombre 27 réalise pas moins de 111 itérations. Le nombre d'itérations ne semble pas avoir de corrélation avec la valeur initiale. Et en cherchant du coté des valeurs maximales atteintes par une suite, que peut-on trouver? Modifions SUITE3 pour cette nouvelle recherche:

```
VARIABLE VALEUR-MAXIMUM
: SUITE4 ( n ---)
 0 VALEUR-MAXIMUM ! \ mise à zéro de la valeur maximale
 1 #ITERATIONS !
 \ initialisation du nombre d'itérations
 BEGIN
    PROCHAIN                  \ calcule nombre de la suite
    DUP VALEUR-MAXIMUM @ >
    \ détermine si n supérieur à val. maxima
    IF  DUP VALEUR-MAXIMUM !
    \ si oui, modifie valeur maximale
    THEN
    DUP 1 = NOT  \ répète si pas égal à 1
 WHILE
    1 #ITERATIONS +!
    \ incrémente compteur d'itérations
 REPEAT DROP ;

: REPETE-SUITE4 ( déb fin ---)
 1+ SWAP
 DO CR ." SUITE DE " I 5 .R ." : "
 \ affiche texte "SUITE DE n: "
    I SUITE4   \ recherche du nombre d'itérations
    #ITERATIONS @ . ." itérations"
    \ affiche "n itérations"
    ."  maxima: " VALEUR-MAXIMUM @ .
    \ affiche valeur maximale
 LOOP ;

 2 50 REPETE-SUITE4    affiche

 SUITE DE     1: 3 itérations maxima: 4
 SUITE DE     2: 1 itérations maxima: 1
 SUITE DE     3: 7 itérations maxima: 16
 SUITE DE     4: 2 itérations maxima: 2
...etc...
 SUITE DE    48: 11 itérations maxima: 24
 SUITE DE    49: 24 itérations maxima: 148
 SUITE DE    50: 24 itérations maxima: 88
```

Le tableau I donne le nombre d'itérations et les valeurs maximales pour toutes les valeurs comprises entre 2 et 199. On aurait pu s'arrêter beaucoup plus loin. Mais pratiquement 200 valeurs traitées permettent déjà de dégager certaines conclusions:

- les puissances de deux constituent des pièges fatals.
- pour l'intervalle étudié, aucun nombre n'échappe à la regression.
- la suite de nombres issue de 27 atteint la valeur maximale la plus élevée (9232).
- certains nombres développent un nombre d'itérations identique et passent par la même valeur maximale. C'est le cas des valeurs consécutives 107 à 111. Pourtant, au départ de la suite, ces valeurs ne commencent pas par le même développement. Faut-il y voir un simple hasard?

Quelle sera la valeur initiale dont le sommet sera supérieur à 9232. Ne cherchez pas, c'est 703 dont la suite de SYRACUSE passe par un maximum de 250504. Une telle valeur ne peut plus être traitée avec des entiers 16 bits. En effet, si le programme de traitement numérique dépassait la barrière de 32767, FORTH considérerait ensuite ces valeurs comme négatives et le programme bouclerait perpétuellement. Il faut donc envisager le traitement de valeurs beaucoup plus importantes par d'autres méthodes.

En traitant des valeurs 32 bits non signées, nous portons la capacité de traitement à 4294967295. C'est ce dont sont chargées d'exécuter les routines suivantes:

```
: UD+ ( ud1 ud2 --- ud3; ud3=ud1+ud2)
  D+ ;
: UD3*1+ ( ud1 --- ud2; ud2=ud1*3+1)
  2DUP 2DUP D+ D+ 1. D+ ;
: UD2/   ( ud1 --- ud2; ud2=ud1/2)
  SWAP 0 D2/ DROP
  \ calcul division par 2 de partie poids faible de ud1
  SWAP DUP 1 AND
  \ cherche si bit poids faible de partie poids fort
  >R             \ =1 et stockage sur pile retour
  0 D2/ DROP SWAP
  \ calcul division par 2 de partie poids fort de ud1
  R>  \ rapp. bit poids faible stocké sur pile retour
  IF  32768 OR THEN
  \ si pas nul, mise à 1 bit poids fort de partie
  SWAP ; \ poids faible de ud1
: D-PROCHAIN ( ud1 --- ud2)
  \ ud2=ud1*0.5 si ud1 pair; ud2=ud1*3+1 si ud1 impair
  OVER 2 MOD
  IF    UD3*1+
  ELSE  UD2/   THEN ;
: D-SUITE ( ud ---)
  BEGIN
     D-PROCHAIN 2DUP UD.     \ affiche nombre de la suite
     KEY UPC ASCII 0 = NOT  \ répète si appui sur 0
  UNTIL ;
```

Et maintenant, on peut essayer sans risque des suites à

partir de nombres très grands. N'oubliez pas le point accompagnant le nombre à traiter:

```
70001. D-SUITE      affiche la suite de SYRACUSE de 70001
```

Pour développer plus rapidement une suite, réécrivons les opérations élémentaires de traitement en code machine:

```
CODE UD3*1+ ( ud1 --- ud2; ud2=ud1*3+1)
  ax pop   dx pop   ax bx mov   dx cx mov
  cx dx add   bx ax adc   cx dx add   bx ax adc
  1 # cx mov   0 # bx mov   cx dx add   bx ax adc
  2push END-CODE
CODE UD2/ ( ud1 --- ud2; ud2=ud1/2)
  ax pop   dx pop   ax shr   dx rcr   2push  END-CODE
```

Et on combine les opérations de développement de la suite de SYRACUSE pour un nombre double précision non signé

```
: D-PROCHAIN2 ( ud1 --- ud2)
  \ ud2=ud1*0.5 si ud1 pair; ud2=ud1*3+1 si ud1 impair
  OVER 2 MOD  IF    UD3*1+   ELSE  UD2/   THEN ;
```

```
2VARIABLE D-VALEUR-MAXIMUM
: D-SUITE ( ud ---)
  CR ." SUITE DE " 2DUP UD. CR
  0. D-VALEUR-MAXIMUM 2!
  \ mise à zéro de la valeur maximale
  1 #ITERATIONS !   \ init. nombre d'itérations
  BEGIN
     D-PROCHAIN2 2DUP UD. \ calcul nombre et l'affiche
     2DUP D-VALEUR-MAXIMUM 2@ D>
     \ si n supérieur à val. maxima
     IF  2DUP D-VALEUR-MAXIMUM 2!
     \ modifie valeur maximale
     THEN
     2DUP 1. D= NOT \ répète si pas égal à 1
  WHILE
     1 #ITERATIONS +! \ incrémente compteur d'itérations
  REPEAT 2DROP  \ puis affiche compteur et val. max.
  CR ." Valeur maximale    : " D-VALEUR-MAXIMUM 2@ UD.
  CR ." Nombre d'itérations: " #ITERATIONS @ . CR ;
```

Tableau I

NOMBRES D'ITERATIONS ET VALEURS MAXIMALES

val ini	nb. iter	maxi mum	val ini	nb. iter	maxi mum	val ini	nb. iter	maxi mum	val ini	nb. iter	maxi mum
			50	24	88	100	25	88	150	15	340
			51	24	232	101	25	304	151	15	1024
2	1	1	52	11	40	102	25	232	152	23	88
3	7	16	53	11	160	103	87	9232	153	36	520
4	2	2	54	112	9232	104	12	52	154	23	232
5	5	16	55	112	9232	105	38	808	155	85	9232
6	8	16	56	19	52	106	12	160	156	36	304
7	16	52	57	32	196	107	100	9232	157	36	472
8	3	4	58	19	88	108	113	9232	158	36	808
9	19	52	59	32	304	109	113	9232	159	54	9232
10	6	16	60	19	160	110	113	9232	160	10	80
11	14	52	61	19	184	111	69	9232	161	98	9232
12	9	16	62	107	9232	112	20	56	162	23	244
13	9	40	63	107	9232	113	12	340	163	23	736
14	17	52	64	6	32	114	33	196	164	111	9232
15	17	160	65	27	196	115	33	520	165	111	9232
16	4	8	66	27	100	116	20	88	166	111	9232
17	12	52	67	27	304	117	20	352	167	67	9232
18	20	52	68	14	52	118	33	304	168	10	84
19	20	88	69	14	208	119	33	808	169	49	4372
20	7	16	70	14	160	120	20	160	170	10	256
21	7	64	71	102	9232	121	95	9232	171	124	9232
22	15	52	72	22	52	122	20	184	172	31	196
23	15	160	73	115	9232	123	46	628	173	31	520
24	10	16	74	22	112	124	108	9232	174	31	592
25	23	88	75	14	340	125	108	9232	175	80	9232
26	10	40	76	22	88	126	108	9232	176	18	88
27	111	9232	77	22	232	127	46	4372	177	31	532
28	18	52	78	35	304	128	7	64	178	31	304
29	18	88	79	35	808	129	121	9232	179	31	808
30	18	160	80	9	40	130	28	196	180	18	136
31	106	9232	81	22	244	131	28	592	181	18	544
32	5	16	82	110	9232	132	28	100	182	93	9232
33	26	100	83	110	9232	133	28	400	183	93	9232
34	13	52	84	9	64	134	28	304	184	18	160
35	13	160	85	9	256	135	41	916	185	44	628
36	21	52	86	30	196	136	15	68	186	18	280
37	21	112	87	30	592	137	90	9232	187	44	952
38	21	88	88	17	52	138	15	208	188	106	9232
39	34	304	89	30	304	139	41	628	189	106	9232
40	8	20	90	17	136	140	15	160	190	106	9232
41	109	9232	91	92	9232	141	15	424	191	44	4372
42	8	64	92	17	160	142	103	9232	192	13	96
43	29	196	93	17	280	143	103	9232	193	119	9232
44	16	52	94	105	9232	144	23	72	194	119	9232
45	16	136	95	105	9232	145	116	9232	195	119	9232
46	16	160	96	12	48	146	116	9232	196	26	148
47	104	9232	97	118	9232	147	116	9232	197	26	592
48	11	24	98	25	148	148	23	112	198	26	448
49	24	148	99	25	448	149	23	448	199	119	9232

Compiling Prolog to Forth

L. L. Odette

Applied Expert Systems, Inc.
5 Cambridge Center
Cambridge, MA 02142

Abstract

The fact that the focus of a Prolog computation is the structure of the program leads directly to a view of a Prolog compiler as a procedure that takes a collection of Prolog clauses and produces a description of their structure that just happens to be executable. Forth lends itself naturally to the description of both structures and processes. In fact, some hold that Forth programming involves creating the parts of speech required to describe an application. This article proposes that for this reason, Forth is a very good language for prototyping Prolog compilers. A simple object language for a Prolog to Forth compiler is presented and discussed.

Introduction

A narrow definition of logic is the study of the arguments valid by virtue of their structure. Having taken this view, a rule language (mechanical theorem prover) needs two elements: a validation process and an internal representation of the argument structure. Rule languages can be distinguished on the basis of how much structure they admit. Expert-2 [PAR84], for example, is a mechanical theorem prover for propositional logic where complex arguments are described only in terms of the atomic propositions that make them up. The requirements of the internal representation of an atomic proposition are met by a token for the proposition (e.g., a pointer to the string representing the proposition's text). Prolog, on the other hand, is a mechanical theorem prover for predicate logic, which attaches significance to the internal structure of the atomic propositions: for example, the predicate name and the number and structure of its formal parameters.

It follows that an interpreter for Prolog must run its validation process over more complex data structures than those used by an interpreter for propositional logic. This added complexity tends to limit the performance of Prolog interpreters, certainly relative to interpreters for propositional languages. The thrust of Prolog compilation is to combine the validation process and the clause structure, so that the internal representation of each atomic proposition *is* the program that realizes the validation process over the proposition. In essence, a compiled Prolog clause is an executable description of the clause, which is why Forth is an ideal language for implementing and experimenting with Prolog compilers.

This paper introduces a set of Forth words which form the basis of a Prolog Virtual Machine (PVM). The instructions of the virtual machine are of two types: those that alter the flow of control and those that denote the structures in Prolog clauses. Compilation to the virtual machine instructions becomes a simple matter of composing a description of the clause, which can easily be done by hand. Implementation of the virtual machine is a straightforward Forth programming task.

The compiler technology presented here is based on the simple compiler described by Bowen, Byrd, and Clocksin [BOW83]. Code for the compiler (in Prolog) is given in an appendix, as is the Forth code for the virtual machine. The compiler code may be of use to Forth programmers interested in building compilers in Prolog. The Forth code may be of use to anyone interested in incorporating Prolog in Forth applications or experimenting with extensions of the Prolog language. The elegance of the Forth solution to compiling Prolog should be of interest to both Forth and non-Forth programmers alike.

Introduction to Prolog

Prolog is a simple language with a straightforward syntax and program structure (Figure 1).

A Prolog program is a set of procedures
A Prolog procedure is a set of clauses
- each clause is of the form "P :- Q1,Q2, ... Qn."
  read: P is true if
    Q1 is true and
    Q2 is true and ... and
    Qn is true.
- if n = 0 the clause is written as "P."
  read: P is true.

Some terminology:

```
P        :-     Q1,Q2,Q3 .
└───┘   └──┘   └─────────┘ └┘
head    neck      body     foot
```

Figure 1. Prolog at a Glance (I).

Its declarative semantics is also straightforward. Each procedure represents the definition of a predicate. For example, the sex of individuals may be specified by the predicates male and female, as defined by the Prolog clauses:

male(isaac).
male(lot).
female(milcah).

Predicate definitions may be conditional, as in the following clauses:

son(X,Y) :- parent(Y,X),male(X).
grandparent(X,Y) :- parent(X,Z),parent(Z,Y).

The first clause is read as "X is a son of Y if Y is a parent of X and X is male." The second clause is read as "X is a grandparent of Y if X is a parent of Z and Z is a parent of Y." The terms **X**, **Y**, and **Z** in these definitions are logical variables, meaning they reference some unknown individual. The scope of a variable reference is the clause it is used in.

What is unusual about Prolog is its procedural semantics – in particular, the search mechanism underlying procedure invocation (which may result in backtracking) and the means for passing information between procedures via unification (pattern matching). Prolog procedures execute much like Forth or any other conventional language with the exception that any procedure call could possibly invoke more than one procedure or even none at all. The Prolog machinery needs to search through the candidate procedures. One way to visualize Prolog procedure execution is as a search tree, or a proof tree in the case of successful search (Figure 2).

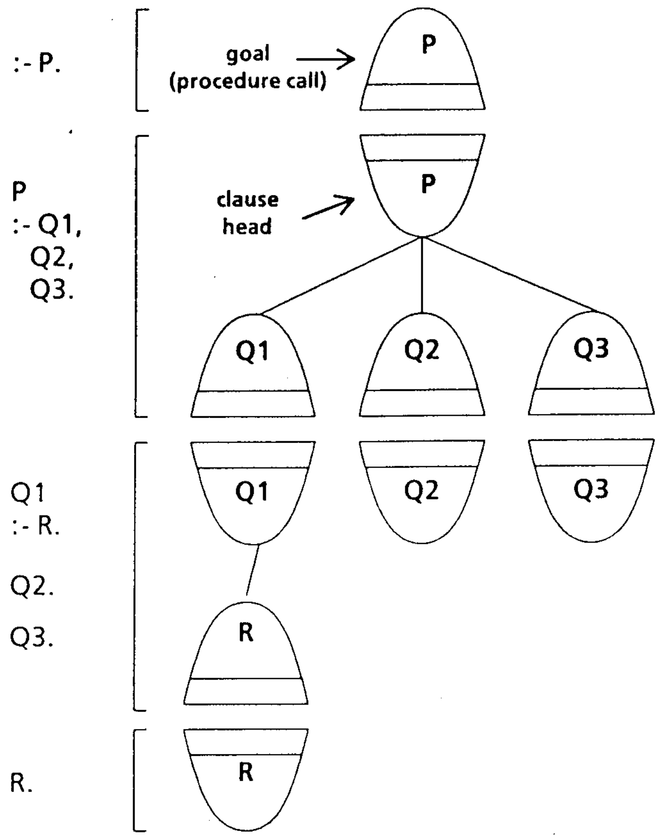


Figure 2. Proof Tree for the Prolog Procedure **P**.

Given the Prolog program on the left, successful execution of the procedure **P**, as invoked by the goal **:- P.** can be represented by the tree on the right. Each upper half circle represents a procedure call while the lower half circle represents a matching procedure. The Prolog machine must search through the program, matching the call against candidate procedures. The expense of the search and the associated pattern matching limit the Prolog performance. (The tree diagram has been called a Ferguson diagram [VAN84].)

Prolog procedures can also have parameters (Figure 3). Unlike parameters in conventional languages, Prolog parameters are neither strictly input nor output parameters. Rather, the role played by a parameter depends on the procedure call, and one of the very unusual things about Prolog parameters is that they can be both input and output. This aspect of parameters is a side effect of one of the more interesting of the ideas about computing that have been realized in the Prolog language. The idea is "call by description." Each parameter of a procedure is a description, as is each argument supplied by a procedure call. Descriptions can be more or less general depending on whether they contain variables or not.

On procedure invocation, the argument terms of the caller (the goal) are matched with the parameter terms of the called procedure. The pattern matching process (called unification) tests whether two terms can be matched by binding some of the variables in the terms. In a sense, unification is an attempt to find a view of the two descriptions under which they describe the same thing. In Prolog, a successful unification of two terms results in the most general description covered by both original descriptions, which may be a specialization of the originals (Figure 4).

Prolog procedures can have terms as parameters

A term may be:
- **a constant**
- **a variable**
- **a structure**

Constants are atomic objects

Variables stand for arbitrary objects
(by convention variable names begin with an uppercase letter)

Structures consist of a functor applied to terms as arguments
(eg. "p(a,b)")

Some terminology:

$\underbrace{\text{p}}_{\text{functor}} \quad \underbrace{\text{(a,b)}}_{\text{arity = 2}}$

Figure 3. Prolog at a Glance (II).

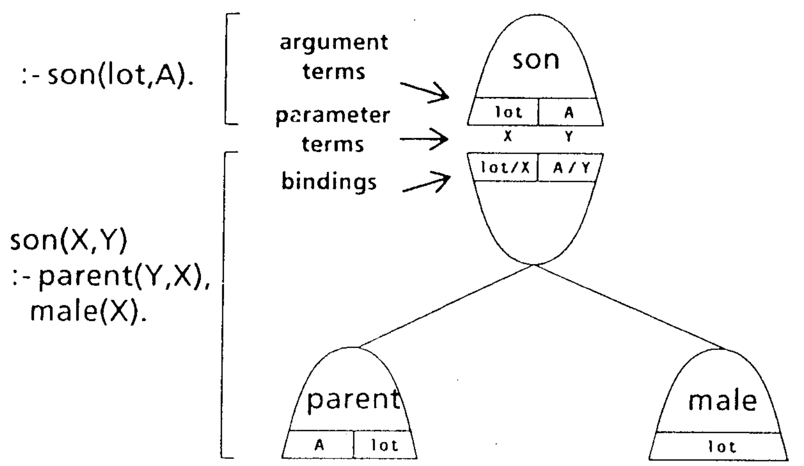


Figure 4. Procedure Invocation by the Goal **:- son(lot,A).**

Read the bindings **a/b** as "a is substituted for b." Following the first successful unification of the goal with the head of the procedure, the variable **X** in the procedure has been specialized to the constant **lot**. The variable A in the goal may be specialized by subsequent unification of the subgoals with other procedures.

The pattern matching procedure involved in unification can be expensive, primarily because so many cases need to be considered (Figure 5).

	Constant Cp	Variable Xp	Structure Sp
Constant Ca	Succeed Ca = Cp	Succeed Xp = Ca	Fail
Variable Xa	Succeed Xa = Cp	Succeed Xa = Xp	Succeed Xa = Sp
Structure Sa	Fail	Succeed Xp = Sa	Succeed if *

* Sa,Sp have same functor and arity
corresponding arguments of Sa,Sp unify

Figure 5. Cases Considered by the Unification Procedure.

Subscripts refer to the arguments passed by the caller (e.g., a structure Sa) and the parameters of the procedure (e.g., a structure Sp). Variables are de-referenced prior to comparisons, meaning if a variable has been bound, it is replaced by the bound value prior to comparison. Unification may recurse on structures.

When the structure analysis done by unification is delayed until run time, as in an interpreter, performance suffers, and, as was pointed out previously, searching for candidate matching clauses on a procedure call can also be expensive. These two observations lead to the basic compilation strategy for Prolog:

Strategy for Compiling Prolog

1) Specialize unification for each clause.

- unification involves an analysis of structure, so move as much of the analysis as possible from run-time to compile-time.

2) Reduce the set of candidate clauses.

- index clauses by their structure. Common indices are main functor, arity and type of first parameter.

The focus of this paper is primarily the implementation of the first strategy because it is relatively easy to see how to approach the implementation of the latter. For example, if all procedures with the same main functor were chained together and accessed through the pfa of the main functor word, there would be a substantial reduction in the search space.

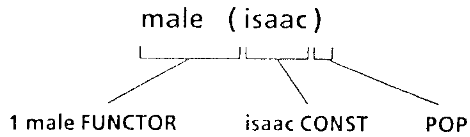
The approach taken here is to break down the process of compiler building into two steps. In the first step, a compiler is described that compiles Prolog to the instruction set of a Prolog Virtual Machine (PVM). The PVM used here has several advantages. First, it is easy to understand and implement because the set of instructions is small (there are only seven instructions). In addition, the compilation procedure is straightforward because there is essentially a one-to-one correspondence between clause structure and the object (PVM) code. Moreover, the PVM is a stack machine, which reduces the complexity of the compiler because issues like register allocation need not be considered. Finally, this PVM serves as a good introduction to the Warren Abstract Machine [WAR83] and the current literature on Prolog compilation.

With the first step being the construction of the compiler, the second step becomes implementing the PVM. This method is a common approach to compiler building, with speed being traded off against the advantages of portability and more compact code.

Prolog Compilation Step 1. Compile Prolog to PVM Instructions

In explaining the compilation procedure, we delay considering full program compilation and look first at the compilation of Prolog structures. The object of the compilation is ultimately to compose a description of the Prolog structure using Forth words, which also happen to be instructions of the PVM. The descriptive words that are needed are the names of the types of Prolog terms – the unstructured terms such as variables and constants, and the structured terms like lists. The term types suggest using PVM instructions named `VAR` and `CONST` for unstructured terms and `FUNCTOR` for structured terms, with an instruction like `POP` used to indicate the termination of a structured term description. These instructions will eventually be implemented as Forth words.

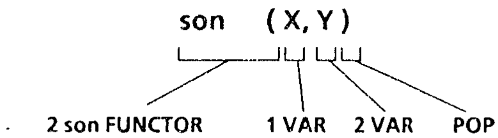
Given these four instructions, the procedure for compiling Prolog structures to PVM instructions is simply to compose a description of the structure using `CONST`, `VAR`, `FUNCTOR` and `POP`. Two examples of the compilation of Prolog structures follow. What is important to note is the near one-to-one correspondence between the Prolog objects that comprise the structure and the PVM instructions the structure compiles into.



Example 1. Compilation of the Structure **male(isaac)**.

The PVM code describes the term **male(isaac)** as a structure with functor = male and arity = 1, whose single formal parameter is a constant = isaac. The PVM instruction `POP` terminates the description. Parameters to PVM instructions are indicated in the familiar Forth reverse Polish syntax.

For a technical reason, the object code does not reference logical variables by their names in the source code. The reason is that variables occurring in a clause must be unique to each use of the clause. Thus, there can be no unique reference to the variable "X." With each procedure invocation, new procedure variables are created and associated with the procedure's stack frame. The compiler renames variables as they appear in a clause – first variable, second variable, etc. – to be used as indices into the area allocated for a procedure's variables. Thus, variables are referenced by number (index) in the object code.



Example 2. Compilation of the Structure **son(X,Y)**.

The PVM code describes the term son(X,Y) as a structure with functor = son, arity = 2. The two formal parameters of the structure are variables referenced by an index into an array of variables.

The procedure for compiling Prolog programs is quite similar to the procedure outlined for compiling structures; however, there are some additional steps and some subtleties. The chief additional step is marking transfer of control via the PVM instructions `CALL`, `ENTER`, and `RETURN`. The chief subtlety is the difference between the way the PVM instructions operate in the head and the body of a clause. In the head of a clause, the PVM instructions perform the operations of unification, as specialized for that clause. In the body of a clause, PVM instructions must prepare arguments for a procedure call. In other words, PVM instructions must operate in at least two modes: "match" mode in the head of a clause and "arg" mode in the body. This fact previews some implementation issues. The reason for mentioning it here is to explain the motivation behind the different forms of description used in the head and body of a clause.

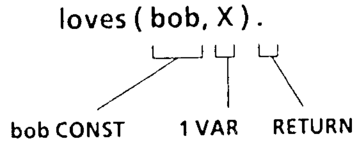
A second subtlety is the effect of clause indexing on the compilation of the head of a clause. It is assumed here that clauses can be indexed by their main functor and arity: if the procedure **son/2** (i.e., functor = son, arity = 2) is being invoked, then candidate clauses can be found by looking, say, at the pfa of the word `son` and following a chain of pointers to the son/2 clauses. The whole Prolog program does not need to be searched, and the functor and arity of the clause can be left off the description of the clause head.

With these additional facts in mind, we first consider the compilation of Prolog clauses without bodies – unit clauses.

Compilation of Unit Clauses

The chief differences between the compiled forms of structures and unit clauses are the indication of transfer of control in the Prolog program with the word `RETURN` and the fact that the functor and arity of the clause are not part of the object code emitted by the compiler. For example,

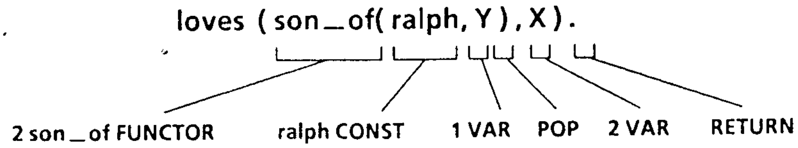
extending the description of the compilation procedure, the clause **loves(bob,X).** is compiled as follows. First the compiler notes that its functor/arity is **loves/2** (this indicates where to store the compiled code) and that it has a single variable **X** and a single constant **bob**. Next a description of the clause is composed as before:



This is the program (description) for **loves(bob,X).** that is stored with the collection of clauses for functor/arity = **loves/2.**

A more complicated example is the clause **loves(son_of(ralph,Y),X).** There are two variables, one constant and a structure in this clause, and the PVM code emitted by the compiler is

```
2 son_of FUNCTOR ralph CONSTANT 1 VAR POP 2 VAR RETURN .
```



Lists may be represented by a structured term with functor/arity = **cons/2** (Figure 6). The first parameter of **cons/2** references the first element of the list, and the second parameter of **cons/2** references the rest of the list. Other representations of lists could be used to save both space and time at the (slight) cost of increasing the PVM instruction set.

External Form	Internal Form
[]	nil
[a]	cons(a,nil)
[a|[]]	cons(a,nil)
[a,b]	cons(a,cons(b,nil))
[a|[b]]	cons(a,cons(b,nil))
[a|b]	cons(a,b)

Figure 6. Prolog List Syntax.

Prolog has several syntactic forms for lists. Generally, a list is enclosed by square brackets. The empty list [ ] is a constant, and the character | separates the beginning of a list from the rest of the list. There is a single internal representation of a list that, in the examples here, is a structure of functor = cons, arity = 2.

As a final example of the compiled form of a unit clause, consider **append([a,b],L,[a,b | L]).** This clause has only one variable. The PVM code emitted by the compiler is

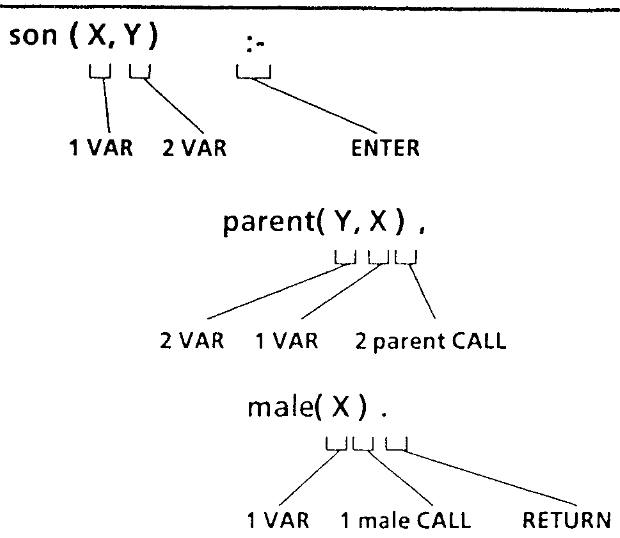
```
2 cons FUNCTOR a CONST 2 cons FUNCTOR b CONST nil CONST POP POP
   1 VAR
2 cons FUNCTOR a CONST 2 cons FUNCTOR b CONST 1 VAR POP POP
   RETURN
```

The compiled code can be read as a description of the structure of the clause **append([a,b],L,[a,b | L]).** With the PVM instruction set implemented in Forth, the description will constitute the program that is executed when **append/3** is called.

Compilation of Non-Unit Clauses

Compilation of non-unit clauses requires two additional PVM instructions: `ENTER` and `CALL`. The word `ENTER` is the object code representation of the "neck" (:-) of a clause. Its chief purpose is to switch the PVM execution mode and adjust certain pointers. The PVM instruction `CALL` takes a reference to the clause to be called as its argument. Its purpose is to transfer control to the called procedure and to save control information.

`CALL` is compiled following a description of the procedure arguments. As mentioned earlier, the compilation of a call is slightly different from the compilation of a structure. For example, consider the clause **son(X,Y) :- parent(Y,X),male(X).** The head of the clause is compiled as for unit clauses, the neck of the clause is marked in the object code by the instruction `ENTER`, and a procedure call is compiled after a description of the arguments to the procedure.



As a final example, consider the clause **append([X | L1],L2,[X | L3]) :- append(L1,L2,L3).** The clause has four variables. The PVM code for this clause is

```
2 cons FUNCTOR 1 VAR 2 VAR POP
  3 VAR
2 cons FUNCTOR 1 VAR 4 VAR POP
  ENTER
2 VAR 3 VAR 4 VAR 3 append CALL RETURN
```

Prolog Compilation Step 2: Implement the Prolog Machine

Once the instructions of the Prolog machine have been named, and it is clear how to compile Prolog clauses to Prolog machine code, what remains is the implementation of the machine. This section describes the simulation of the Prolog machine in software.

There are three main components of the simulation. The first is the internal representation of Prolog terms (e.g., constants, variables, and structures) and of references to these objects. The second component concerns the structure of the stacks required to support Prolog computation. The final component is the procedural semantics of the PVM instructions – what the instructions do. Side issues like the memory map, implementation registers and scratch stacks will be touched on but in less depth.

Internal Representation of Terms and References to Terms

First consider references to Prolog terms. As previously mentioned, there are three primitive types of Prolog terms. One internal representation could be a 32-bit cell with the 2 high-order bits indicating the type of the term and the remaining bits containing a pointer to the term (this is just a generalization of the idea of pointer). The two fields of the reference are called the "tag" and the "val," following Clocksin [CLO85]. The following represents a sufficient set of references to primitive objects:

Tag	Val	Purpose
1	pointer to a variable binding	variable
2	pointer to a constant record	constant
3	pointer to a structure record	structured term

For the sake of efficiency, it may be desirable to increase the number of types of terms that can be referenced. For example, it can be worthwhile to have a special type of reference for integers even though integers could very well be referenced like any atom. Similarly, one might wish to reference lists as distinct from general structures and unbound variables as distinct from bound variables.

The simplest of the internal representations of terms is the representation of variables. Variables and references to variables are identical. The val field of a variable reference points to a reference to a Prolog term. An unbound variable is often indicated by a reference structure that has the tag field of a variable and a val field that points to itself.

The internal representations of Prolog constants and structures are distinct from the representations of references to these types of terms. Both constants and structures are represented by different kinds of records, with distinct fields holding relevant information about the term. For example, the record representing a structure holds the information about its functor and arity, as well as references to its parameter terms.

Constant and structure representations are built in different areas of memory. Structures reside exclusively in an area of memory called the structure stack. This stack constitutes the necessary dynamic memory allocation required for Prolog computation and simplifies the garbage collection problem because stacks grow and shrink with the computation.

Constants also reside in a special area of memory. For a simple Forth implementation of Prolog, we can identify the Forth dictionary with the constant space (presumably Prolog would exist in a separate Forth vocabulary). Each Prolog constant is then represented by a Forth word, the header storing the name string and the parameter field storing other information. This particular implementation scheme leaves the garbage collection of constants unresolved, which may be a problem.

The Forth dictionary can also be the place where Prolog programs are stored. Indexing of clauses in a particular procedure can be done through the main functor of the procedure. One simple way to do this is to chain procedures by arity, with the pointer to this chain stored in the pfa of the main functor word. Clause records can then be chained off the procedure records. With this structure, procedure invocation begins with a search down the procedure links, and backtracking resumes a search down the clause links.

Garbage collection of procedures may be necessary if there is significant data base manipulation in a Prolog program. This could be accommodated here by allocating space for procedures from a heap [DRE85].

In summary, there are four kinds of record structure:

Constant record (2 fields). The first field is the name string of the constant, and the second field is a pointer to a chain of procedure records. For our purposes, a constant record is a Forth word, the header containing the name string, link field, etc., and the first cell of the parameter field pointing down the procedure chain (i.e., the Forth code used to build and initialize constant records is `CREATE 0 ,`).

Structure record (3 fields). The first field is a pointer to the constant naming the functor, the second field holds the number of arguments of the structure, and the third is a variable length field containing the references to the formal arguments of the structure. Prolog structure records are built by `FUNCTOR` descriptions in space allocated from the structure stack.

Procedure record (3 fields). The first field is a pointer to the next procedure in the chain having the same functor but different arity, the second field holds the arity of the procedure, and the third field contains a pointer to a chain of clause records. Space for procedure records can be allocated from the Forth dictionary or from a heap.

Clause record (3 fields). The first field is a pointer to the next clause record, the second field holds a number indicating how many variables are in the clause, and the third is a variable length field that contains the code itself (effectively the Forth parameter field). Space for clause records can be allocated from the Forth dictionary or from a heap.

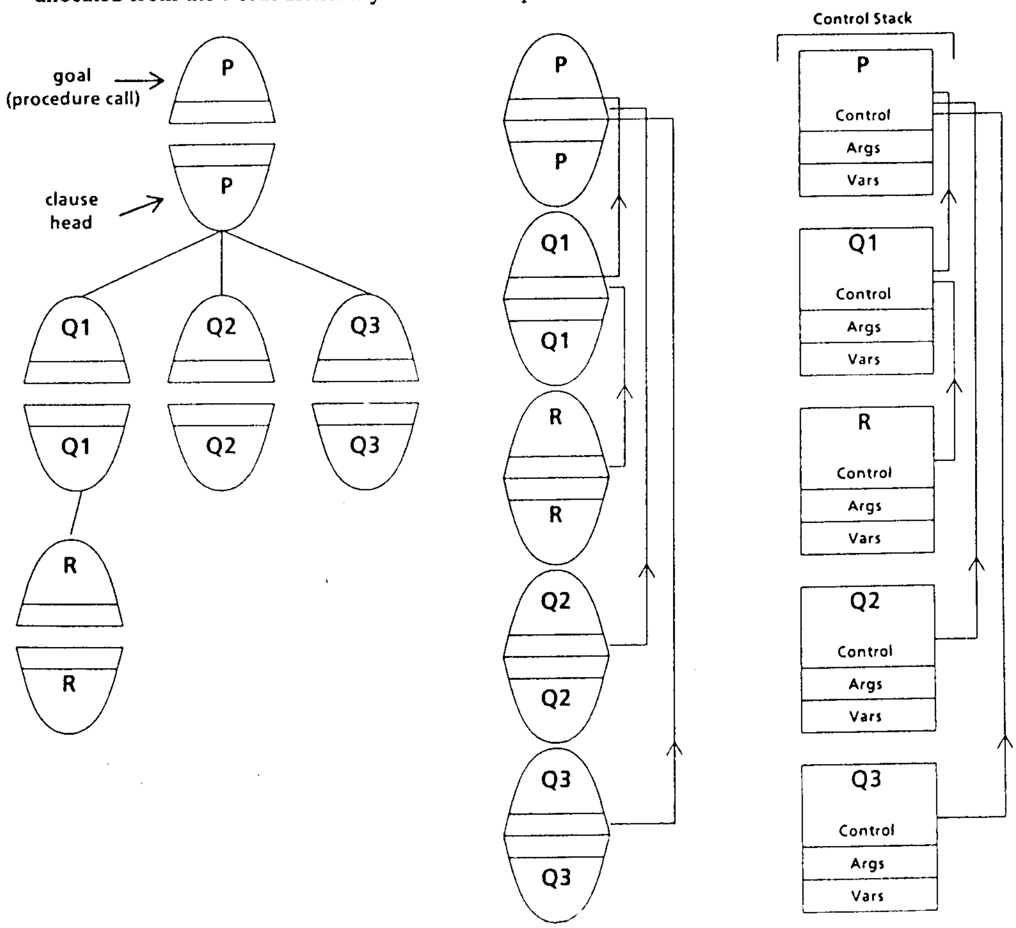


Figure 7. Structure of the Control Stack.

The control stack (right-hand side) constitutes a trace of the procedure calls during a Prolog computation and therefore is a representation of the Prolog proof tree (left-hand side), with each stack frame corresponding to a procedure call. The stack frame holds control information, procedure arguments and the clause variables. In practice, it is possible to reclaim space on the control stack during a computation.

Stack Structure

There are two main stacks in a Prolog machine. The first is the structure stack that holds any temporary structures created during the computation. This is a straightforward stack requiring only a pointer to its top. The second of the Prolog stacks (control stack) holds state information, the arguments passed to procedures and procedure variables. This stack is essentially a linear version of the proof tree traced out during a Prolog computation (Figure 7).

The Prolog stacks must generally be large relative to the usual Forth stacks because, in the case of the structure stack, the stack is the mechanism for dynamic memory allocation, and, in the case of the control stack, nondeterminism requires that all state information be saved in case backtracking is necessary. Thus, a procedure return does not necessarily pop the control stack, and the stack can grow quite deep.

Implementation of PVM Instructions

The following section describes in detail the workings of software simulations of the PVM instructions. The first issue is execution modes. As alluded to previously, the PVM instructions `CONST`, `VAR` and `FUNCTOR` operate in modes, the two main modes being "match" and "arg." There is a third mode called "copy" that is a variant of "arg" mode.

The instructions operate in match mode in the head of a clause, matching the parameters of the clause with the arguments passed to the procedure on the control stack. Instructions operate in arg mode in the body of a clause, placing arguments on the control stack prior to a procedure call. Modes are switched by the instructions `CALL`, `ENTER` and `RETURN` (Figure 8).

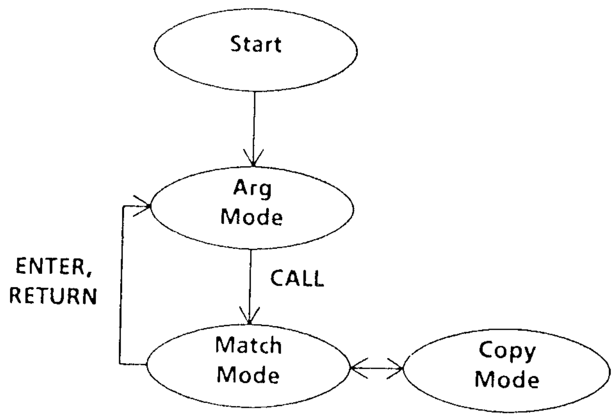


Figure 8. Mode Switching in the Prolog Machine.

A procedure invoked from top level would begin executing in arg mode, placing arguments on the control stack prior to a call. The call (PVM instruction `CALL`) switches the mode to match, and the arguments are matched with the parameters in the head of the clause. If the match is successful, the body of the clause is entered (PVM instruction `ENTER`), the mode is switched to arg, and arguments are placed on the control stack prior to the first call in the body. The mode is also switched to arg on a procedure return (PVM instruction `RETURN`). This is only strictly necessary when returning from unit clauses.

`CONST`, `VAR` and `FUNCTOR` in Arg Mode

In the discussion to follow, operation of PVM instructions in each mode will be considered according to the mode sequence pictured in Figure 8 – first arg mode, then match mode, and finally copy mode. To begin, we look again at how a procedure call (goal) compiles, focusing now on what the code does. For example, the goal **parent(haran,X)** compiles to the following PVM instructions.

```
haran       CONST      % push reference to haran
n           VAR        % push reference to variable
2 parent    CALL       % call parent/2
```

If this procedure call is successful, the variable **X** is bound to the child of **haran**.

At this stage of execution, the PVM is in arg mode, and the effect of PVM instructions is to place references to arguments on the control stack. An argument pointer is maintained to indicate where the arguments are to go. The actions of PVM instructions in arg mode are described as follows.

Instruction	Parameter(s)	Description (arg mode)
CONST	C: pointer to a atom	push reference to C on control stack advance arg pointer, continue
VAR	I: index into environment	dereference Ith variable push result on control stack advance arg pointer, continue
FUNCTOR	F: pointer to an atom N: integer	build F/N on structure stack push reference to it on control stack, push copy of arg pointer, reset arg pointer to 1st parameter of F/N, continue
POP	none	pop arg pointer, continue

Thus, in executing the goal **parent(haran,X)**, the control stack has two argument references on it, the argument pointer indicating the first of these, just before the procedure call is made (Figure 9).

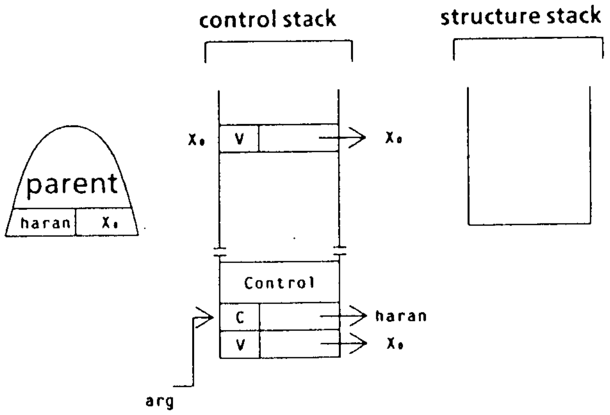


Figure 9. Stacks before `CALL`.

References to the arguments have been loaded on the control stack, and an argument pointer is set to the first argument reference. The `C` in the tag field of the first argument indicates that it references a constant; its val field points to the constant. The `V` in the tag field of the second argument indicates that it references a variable; its val field points up earlier in the control stack to the original variable reference. The fact that the val field of this earlier variable reference points to itself denotes that the variable is unbound.

The PVM instruction sequence `2 parent CALL` results in a search through the procedure records from **parent**, looking for procedures whose arity is 2. If one is found, the execution mode is switched to match, control is transferred to the procedure code, and the pattern matching process begins. Transfer of control instructions are detailed later.

A more complicated example, one that involves the structure stack, is the code for the goal **derivative(sin(a),a,Y)**. If this procedure is successful, it results in the binding of the variable **Y** to the derivative of **sin(a)** with respect to **a**.

```
1 sin          FUNCTOR % create sin/1, push reference, reset arg
                         pointer
a              CONST   % push reference to a
               POP     % restore arg pointer to derivative/3 from
                         sin/1
a              CONST   % push reference to a
n              VAR     % de-reference var, push reference
3 derivative CALL
```

Thus, in executing this goal, the control stack has three argument references on it just before the procedure call is made, and the argument pointer indicates the first of these (Figure 10). The first argument references a structure on the structure stack.

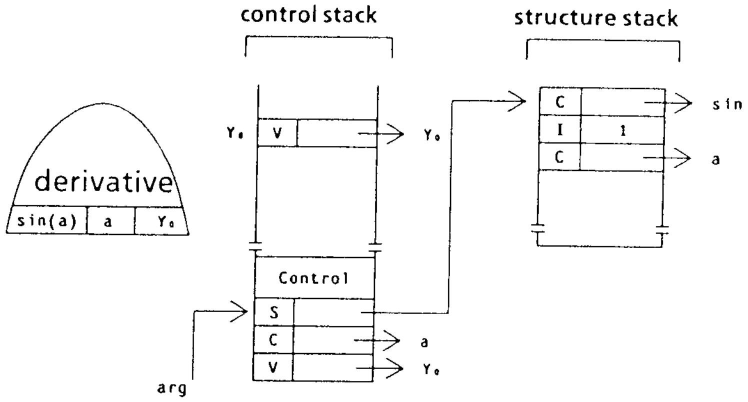


Figure 10. Stacks before CALL.

References to the arguments have been loaded on the control stack, and an argument pointer is set to the first argument reference. The S in the tag field of the first argument indicates that it references a structure; its val field points to the structure, which is located in the structure stack. The structure was built by the sequence of instructions `1 sin FUNCTOR a CONST POP` operating in arg mode.

As before, the PVM instruction sequence `3 derivative CALL` results in a search through the procedure records from the word `derivative`, looking for any procedures whose arity is 3. If one is found, the execution mode is switched to match, control is transferred to the procedure code, and the pattern matching process begins.

CONST, VAR and FUNCTOR in Match Mode

The CALL instruction switches the execution mode of the PVM to match before transferring control. In this mode, the PVM instructions of the compiled form effect the matching between the argument and the parameters of the clause. For any instruction, if the argument is an unbound variable, the instructions immediately bind that variable to the appropriate term. For arguments that are other than unbound variables, the actions of PVM instructions in match mode are described as follows.

Instruction	Parameter(s)	Description (match mode)
CONST	C: pointer to a atom	if arg not a constant, fail else if arg not = C, fail else advance arg pointer, continue
VAR	I: index into environment	dereference Ith variable if result is an unbound variable, bind to arg, advance arg pnter, continue else if result type not = arg type, fail else unify result and argument if unificiation not successful, fail else advance arg pointer, continue
FUNCTOR	F: pointer to an atom N: integer	if arg not a structure, fail else if functor of arg not = F or arity of arg not = N, fail else push copy of arg pointer, reset arg pointer to 1st parameter of arg, continue
POP	none	pop arg pointer, continue

As an example of match mode operation, consider the compiled forms of unit clauses **parent(haran,lot).** and **parent(abraham,isaac).**

```
haran     CONST    % match 1st arg with haran
lot       CONST    % match 2nd arg with lot
          RETURN

abraham   CONST    % match 1st arg with abraham
isaac     CONST    % match 2nd arg with isaac
          RETURN
```

The operation of this code in realizing the unification is straightforward (Figure 11). If any of the matches fail, backtracking is invoked.

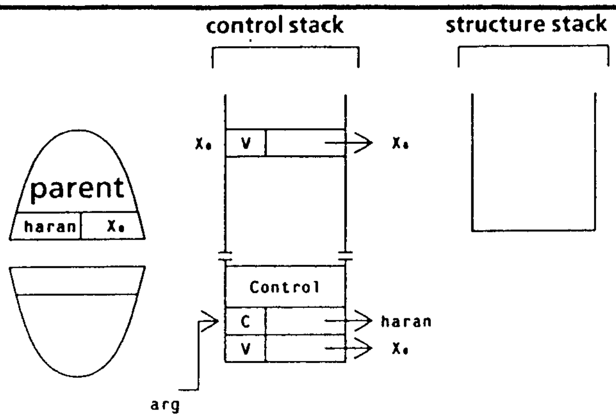


Figure 11a. After `CALL`, before `haran CONST`.

Before the beginning of execution of the PVM code for the procedure, the control stack contains the arguments, and an argument pointer indicates the first of them. The PVM code `haran CONST` will check whether the first argument references the constant **haran**. In the case illustrated, the first argument does reference **haran** so the match succeeds, the argument pointer is advanced, and execution continues.

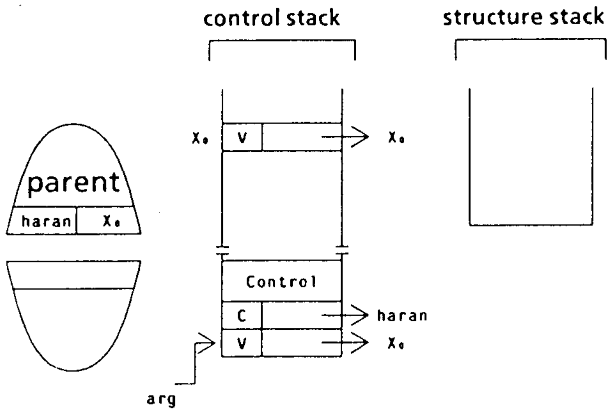


Figure 11b. After `haran CONST`, before `lot CONST`.

The argument pointer now points to the second argument, which references an unbound variable. The PVM code `lot CONST` notes that the argument is an unbound variable and therefore binds it by making it a reference to the constant **lot**.

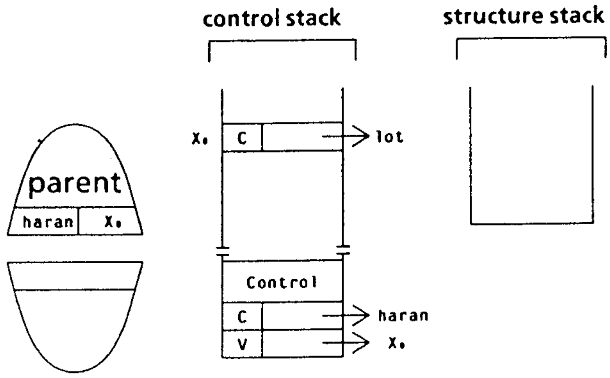


Figure 11c. After `lot CONST`, before `RETURN`.

Note that the variable referenced by the second argument has been replaced by a reference to the constant **lot**. At this point the stack frame for the procedure could be reclaimed if no more alternatives remained. Otherwise argument and control information must be maintained for this procedure in the event that the computation backtracks to this point.

Two additional illustrations of unit-clause PVM code follow. The first example is the code for the clause **derivative(sin(X),X,cos(X)).**, which states that the derivative of the sine of any argument with respect to that argument is the cosine of that argument. This clause compiles to:

```
1 sin  FUNCTOR  % match 1st arg with sin/1, reset arg pointer
1      VAR      % match 1st parameter of sin/1 with first var
       POP      % restore arg pointer to derivative/3 from sin/1
1      VAR      % match 2nd arg with first var
1 cos  FUNCTOR  % match 3rd arg with cos/1, reset arg pointer
```

```
1       VAR       % match 1st parameter of cos/1 with first var
        POP       % restore arg pointer to derivative/3 from cos/1
        RETURN
```

The second example, which contains nested structures, is the clause

derivative((sin(X),2),X,*(2,*(sin(X),cos(X)))).**

This clause states that the derivative of the square of the sine of some argument is twice the product of the sine and the cosine. Using infix notation the clause would read

derivative(sin(X)2,X,2*sin(X)cos(X)).**

The clause compiles to:

```
2 **    FUNCTOR  % match 1st arg with **/2, reset arg pointer
1 sin   FUNCTOR  % match 1st parameter of **/2 with sin/1, reset arg pointer
1       VAR      % match 1st parameter of sin/1 with first var
        POP      % restore arg pointer to **/2 from sin/1
2       CONST    % match 2nd parameter of **/2 with "2"
        POP      % restore arg pointer to derivative/3 from **/2
1       VAR      % match 2nd arg with first var
2 *     FUNCTOR  % match 3rd arg with */2, reset arg pointer

2       CONST    % match 1st parameter of */2 with "2"
2 *     FUNCTOR  % match 2nd parameter of */2 with */2, reset arg pointer
1 sin   FUNCTOR  % match 1st parameter of */2 with sin/1, reset arg pointer
1       VAR      % match 1st parameter of sin/1 with first var
        POP      % restore arg pointer to **/2 from sin/1
1 cos   FUNCTOR  % match 2nd parameter of */2 with cos/1, reset arg pointer
1       VAR      % match 1st parameter of cos/1 with first var
        POP      % restore arg pointer to **/2 from cos/1
        POP      % restore arg pointer to **/2 from **/2
        POP      % restore arg pointer to derivative/3 from **/2
        RETURN
```

CONST, VAR and FUNCTOR in Copy Mode

The remaining complication that must be dealt with in respect to the operation of the PVM instructions CONST, VAR and FUNCTOR is operation in copy mode. Copy mode is entered when an argument is an unbound variable and the corresponding parameter is a structure. In this case, the structure must be built and placed on the structure stack, and the variable reference must be replaced by a reference to the structure. The process of building the structure is similar to what takes place in the body of a clause except that, in this case, the structure building code is in the clause head – thus the need for a different mode. The operation of the PVM instructions in this mode is described in the following table.

Instruction	Parameter(s)	Description (copy mode)
CONST	C: pointer to a atom	copy C reference to structure stack, advance arg pointer, continue
VAR	I: index into environment	dereference Ith variable if result is an unbound variable, create new unbound var on struct. stack bind referenced var to new var else copy reference to structure stack then advance arg pointer, continue
FUNCTOR	F: pointer to an atom N: integer	build F/N on structure stack push copy of arg pointer, reset arg pointer to 1st parameter of struct, continue
POP	none	pop arg pointer, continue

As an example of copy mode operation, consider the clause

derivative(sin(X),X,cos(X)).

as called by

derivative(sin(a),a,Y).

Pictures of the stacks through execution are given in Figure 12.

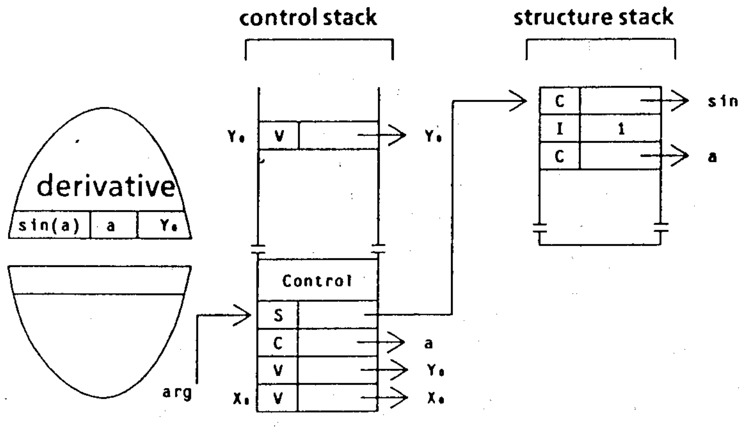


Figure 12a. After `CALL`, before `1 sin FUNCTOR`.

Before the beginning of execution of the PVM code for this procedure, the control stack contains the arguments, and an argument pointer indicates the first of them. Because the clause contains one variable, space has been allocated on the control stack following the procedure arguments, and the variable has been initialized as unbound. The PVM code `1 sin FUNCTOR` will check whether the first argument references a structure with functor = sin and arity = 1. In the case illustrated, the first argument does reference such a structure so the match succeeds, a copy of the argument pointer is saved, and the argument pointer is set to point to the first parameter of the structure.

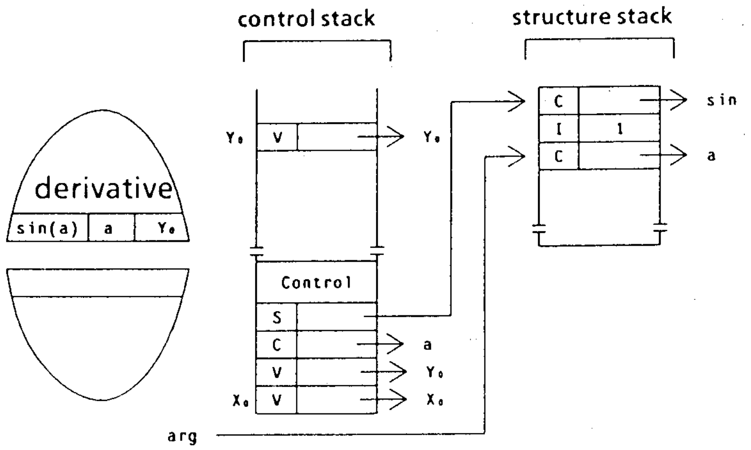


Figure 12b. After `1 sin FUNCTOR`, before `1 VAR`.

The PVM code `1 VAR` will de-reference the procedure's first variable and compare the result with the reference pointed to by the argument pointer. At this stage of the computation, the variable is unbound, and the argument reference is to the constant a. The variable is then bound to a (its reference is changed to a). The argument pointer is advanced.

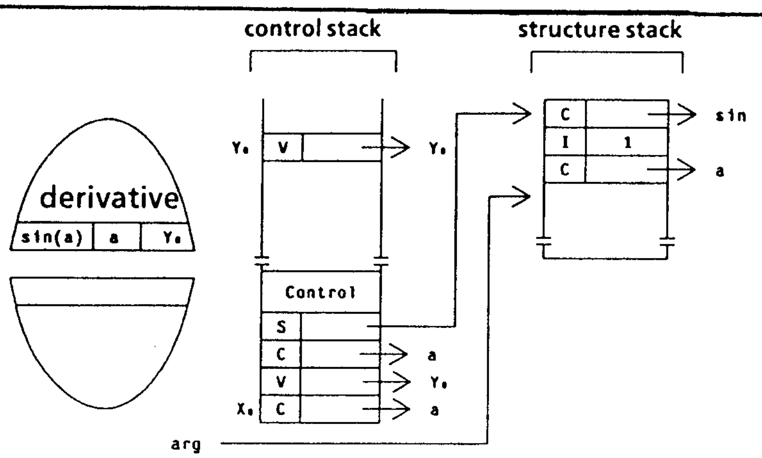


Figure 12c. After 1 VAR, before POP.

The PVM code POP will restore the argument pointer to the value it had before FUNCTOR was executed. Note that the cell allocated for the first variable of the procedure now references the constant a.

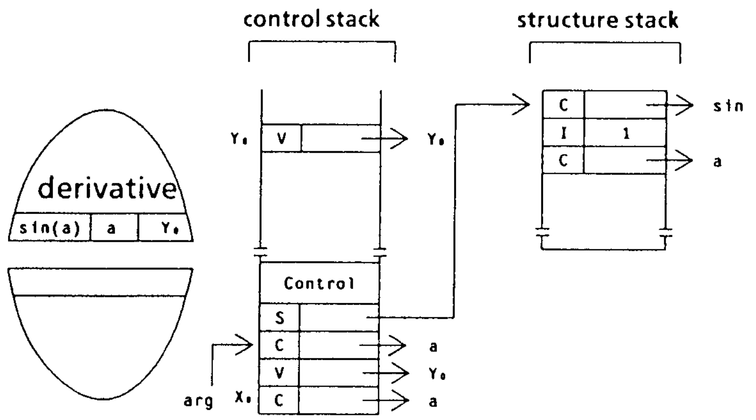


Figure 12d. After POP, before 1 VAR.

The PVM code 1 VAR will consult the term referenced at the memory location of the first procedure variable and compare the result with the reference pointed to by the argument pointer. At this stage of the computation, the variable is bound to the constant a and the argument reference is to the constant a; therefore, the variable and the argument will match. The argument pointer is advanced.

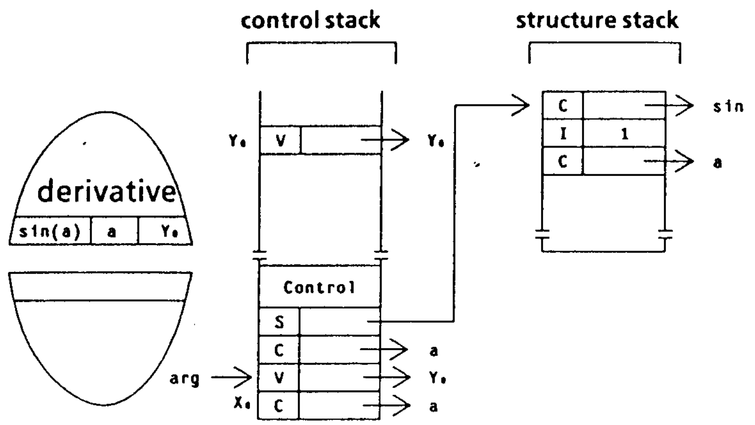


Figure 12e. After 1 VAR, before 1 cos FUNCTOR.

The PVM FUNCTOR instruction will notice that the next argument references an unbound variable. The mode will be switched to copy, and a structure will be constructed on the structure stack. The structure is known to have functor = cos and arity = 1; therefore, space for the structure can be allocated, and the corresponding structure reference can replace the unbound variable reference.

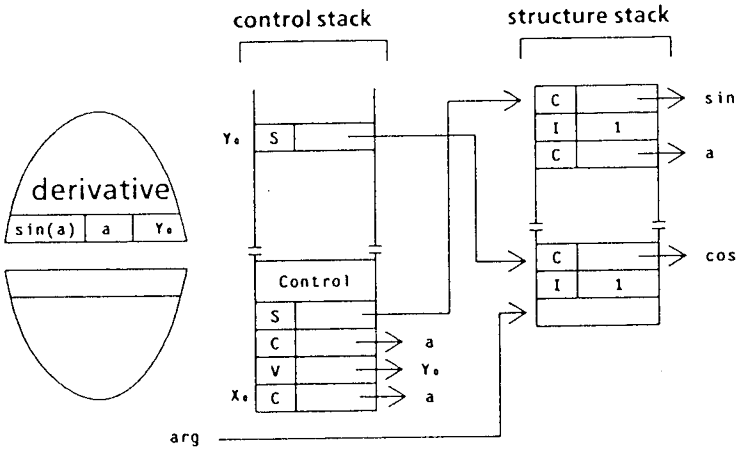


Figure 12f. After 1 cos FUNCTOR, before 1 VAR.

Once space for the structure has been allocated and the variable bound, a copy of the argument pointer is saved, and the pointer is reset to the first parameter position of the new structure. Following PVM code will cause references to Prolog terms to be placed at the positions indicated by the argument pointer.

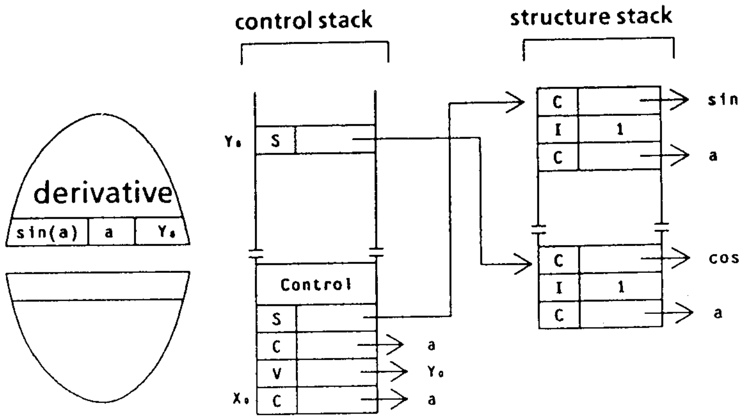


Figure 12g. After 1 VAR POP.

The first variable is again de-referenced and copied to the position indicated by the argument pointer. Execution of POP will restore the argument pointer to its value before execution of FUNCTOR and change the execution mode back to match.

In closing this section, some final comments on references to Prolog terms and the binding of Prolog variables are in order.

- The only Prolog terms that live in the control stack are variables.
- Structures live only in the structure stack. No subterm of a structure exists in the control stack.
- Variables in the control stack can be bound only to constants, terms in the structure stack, or variables occurring earlier in the control stack.

Maintaining this discipline facilitates the restoration of the state of the Prolog computation in case backtracking is required. The fact that structures live completely and only in the structure stack means that they can be readily disposed of on backtracking simply by changing the pointer to the top of the structure stack. Similarly, variable-variable binding is required to be from the most recent variable to the least recent variable, both in the control and the structure stacks. This binding discipline simplifies backtracking and means as well that the control stack frame for deterministic procedures may be reclaimed without creating dangling pointers.

There must also be a mechanism that will note the binding of variables that have been created before the most recent backtrack point because, on backtracking, the bindings of these variables must be undone. The mechanism is a special stack called the trail. On binding a variable that lives earlier than the most recent backtrack point, a pointer to the variable is pushed on the trail. The trail stack pointer is part of the control information saved with a control frame, thereby providing the necessary information to reset variables on backtracking.

Transfer of Control Instructions CALL, ENTER and RETURN

The instructions of the PVM that remain to be described are the flow of control instructions CALL, ENTER and RETURN. Most of what these instructions do has been described previously and is summarized in the following table.

Instruction	Parameter(s)	Description
CALL	F: pointer to an atom N: integer	find first clause with functor F arity N, if found, allocate space for variables, copy control information to control stack, if current clause has remaining alternatives, update backtrack pointer, copy backtrack info to control stack set execution mode to "match" transfer control to clause else fail
ENTER	none	set execution mode to "arg", adjust stack frame pointers
RETURN	none	if deterministic, reclaim control stack frame set execution mode to "arg" transfer control back to caller

As an example of the compilation of a full clause, consider

son(X,Y) :- parent(Y,X),male(X).

which compiles to:

```
1          VAR     % match 1st arg with first var
2          VAR     % match 2nd arg with second var
           ENTER   % set execution mode to arg
2          VAR     % de-reference then copy 2nd var to control stack
1          VAR     % de-reference then copy 1st var to control stack
2 parent   CALL    % transfer control to parent/2 or backtrack
1          VAR     % de-reference then copy 1st var to control stack
1 male     CALL    % transfer control to male/1 or backtrack
           RETURN  % reclaim stack area, return control to caller
```

Several Prolog machine implementation registers are needed to support the computation (Figure 13). These registers contain pointers into the code, pointers to the control, structure and trail stacks, a flag indicating the execution mode, and the argument pointer. Some of the registers are saved by the instructions `CALL` and `ENTER` and then restored by `RETURN` and the backtracking mechanism. `CALL` and `RETURN` always save and restore the program counter and a pointer to the control stack frame of the current procedure. These are the first two registers in the table that follows. If the procedure is deterministic, these are the only two registers saved. If a procedure is non-deterministic – there are remaining alternatives (as indicated by the link on the code record) – `CALL` saves the contents of all six registers in the table. These six constitute sufficient information to restore the execution state on backtracking (Figure 14).

Register	Description
RC	pointer to code; the return point in the calling procedure
RF	pointer to the control stack; stack frame of the calling procedure
BC	pointer to a procedure; next procedure on backtracking
BF	pointer to the control stack; last choice point
SS	pointer to structure stack; reset to this value on backtracking
TS	pointer to trail stack; reset variables on here on backtracking

Figure 13. Prolog State Registers.

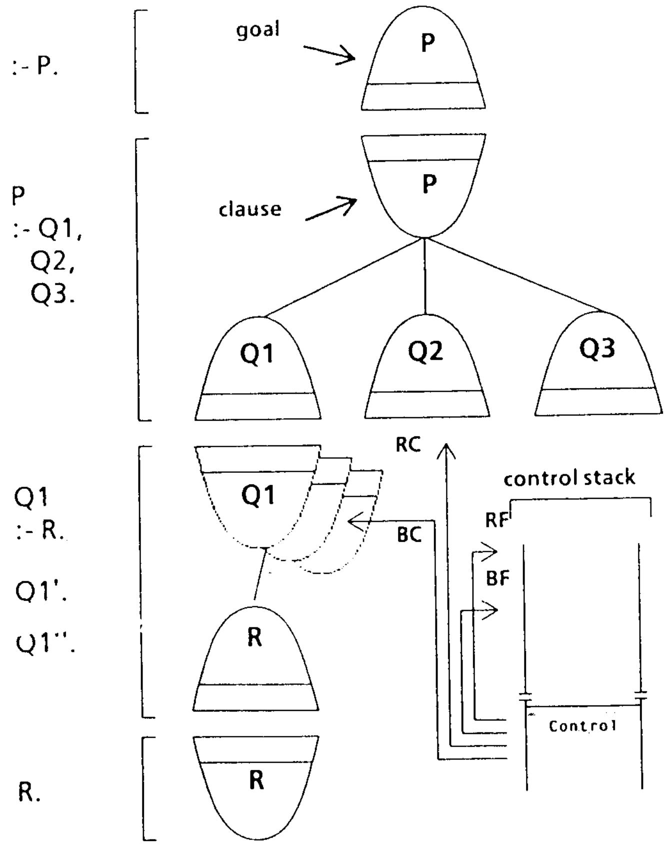


Figure 14. Control Information Saved by CALL.

This figure indicates the main registers saved by CALL in the case of a call to a procedure that has more than one clause. Also saved are the trail and structure stack pointers.

Backtracking

Backtracking occurs on failure to find a procedure with the correct functor and arity (CALL), on failure to match the arguments of a call and the parameters of a procedure (CONST, VAR, FUNCTOR), or on explicit invocation via the predicate **fail**. The following events are triggered by backtracking:

- Go back to most recent choice point (set current frame to contents of BF register).
- If there is only one remaining alternative clause, update the choice point (restore BF from (new) current frame if necessary).
- Garbage collect the structure stack (restore SS from (new) current frame).
- Re-initialize variables where necessary (restore TS from (new) current frame; unbind any trailed variables).
- Transfer control to the next alternative clause (reset program counter from BC in (new) current frame).

PVM Implementation in Forth

A Forth word set that implements the PVM instructions described here is given in Appendix A. This code supplies most of the functionality required by the Prolog machine. The word set described in the appendix has been built in MicroMotion MasterForth and runs on the Apple Macintosh. Using colon words exclusively, the compiled Prolog runs over ten times faster than an earlier Prolog interpreter in Forth [ODE87].

An optimized version of the PVM (see optimization below) has been ported to the NC4000P Forth engine where it runs the naive reverse benchmark at 6K LIPS with a clock rate of 4 MHz. (The Logical Inference Per Second measures, in effect, the procedure call rate.) At a clock rate of 10 MHz., it is estimated that this version would achieve performance equivalent to the fastest compiled Prolog (Quintus Prolog) running on the VAX 11/780 [ODE86].

There are some minor differences between the PVM described here and the version simulated by the Forth code in the appendix. The most significant difference is the interpretation of the argument of the PVM instruction VAR. The Forth code for VAR uses a byte offset from the start of the stack frame to locate a variable. By contrast, the PVM VAR instruction described above uses an index into a table of variables. The latter convention makes the description of the PVM less complicated; the former makes the PVM execution somewhat faster.

With some extra work, the PVM to Forth compiler (the word ASSERTZ in screen 40) could calculate the byte offset from the table index. For example, in a clause with two parameters, the first variable is allocated on the control stack after the control information (12 bytes) and the arguments (2 arguments times 4 bytes/argument), so its offset from the start of the stack frame would be 12 + 8 = 20 bytes.

A second difference between the general PVM and the Forth simulation lies in the way references to Prolog objects are tagged. Since a small model Forth is assumed in the simulation, all pointers are 16 bits, and therefore the high order 16 bits of the object reference are free to be used for the tag. This makes both tagging and testing tags very simple. A version for a large model Forth would require more complicated code for these operations.

The Compiler

Basics

The Prolog compiler whose (Prolog) code appears in Appendix B accepts a restricted Prolog syntax (Figure 15). The most important restriction of the syntax is that all predicates be expressed in functional form. Extension of the compiler to accept other operator positions does require a significant effort, although a straightforward path to the more general syntax would be to build a preprocessor that transforms all predicates into functional form. The output of this program could then be used as the compiler input, and the compiler per se would not have to be modified. There are other parts of the usual grammar that are not recognized by the grammar used here (e.g., strings), but adding them requires only simple modifications.

```
<horn_clause> ::= <atmf>. | <atmf> :- <atmfs>.

<atmfs> ::= <atmf> {,<atmf>}

<atmf> ::= <atom_name> | <atom_name> ( <args> )

<args> ::= <simple_term> | <simple_term> , <args>

<simple_term> ::= <atom_name> ( <args> ) |
                  <variable> |
                  <constant> |
                  <list> |
                  (<simple_term>) |
                  (<conjunction>)

<conjunction> ::= <simple_term> , <simple_term>
                  <simple_term> , <conjunction>

<atom_name> ::= <lower case identifier>

<variable> ::= <identifier starting with uppercase or "_">

<constant> ::= <atom_name> | <integer>

<list> ::= [ <simple_term> ] |

          [ <simple_term> {, <simple_term> } | <list> ]
```

Figure 15. Grammar Accepted by the Compiler.

The input to this compiler is a list of tokens for a single clause, terminated by the token . . The tokens are the names of each constant, variable and functor, along with parentheses, quotes, punctuation and the clause neck (:-). The tokenizer is not described here, but it is relatively easy to construct (see [CLO81], p. 86). Note that the grammar accepted by the compiler recognizes structured terms with spaces between the functor name and the left parenthesis bracketing the functor arguments. One approach to improving on this is to annotate the identifiers produced by the tokenizer, thereby indicating to the compiler that an atom followed immediately by a (is a functor name.

The compiler is implemented as a grammar, using the grammar rule facility provided in most Prologs [CLO81]. The grammar consists of a collection of rules that define the strings of symbols that are valid sentences of the language. Grammar rules may also provide for some analysis of the sentence, often transforming it into a structure which is meant to clarify its meaning. The grammar presented here analyzes the input string in this manner, transforming it into code for the Prolog machine.

Optimizations

The compiler and PVM have been simplified as much as possible for the purpose of exposition; however, there are a number of modifications that will increase execution efficiency (at the expense of increasing the complexity of the compiler and adding words to the Forth vocabulary). For example, the code density could be reduced by putting all object references for each clause into a table. Then, instead of each type word taking an object reference as its argument, it could just take an index into the reference table. Type words could then be specialized by index, e.g., `1CONSTANT`, `2CONSTANT`, etc. The result would be that, in the code, only one cell is required for most primitive object descriptions instead of two. The cost is the time required to extract the references from the table.

The PVM instructions may be specialized in other ways. For example, the constant **nil** could be described by a special word, such as `CONSTNIL`, thereby saving both time and space in the reference table. A special functor description word for **cons/2** is also desirable since lists are a very common structure. Similarly, unnamed variables could be described by a special PVM instruction such as `VOID`.

Specialization of variable descriptions also provides a number of opportunities to increase efficiency. Instead of initializing variables on entry into a procedure, variables could be initialized on first appearance in the clause and compiled to a PVM instruction called, for example, `FIRST.VAR`. In match mode, such a special description would also save the check to determine the binding of a variable. Consecutive unnamed variables might also be compiled to a single word of one argument.

Finally, one might consider combining `CALL-RETURN` pairs into a single description and compiling the cfa of special-purpose functions directly. Directions for further extensions to the word set are suggested in [CLO85] and [WAR83].

Mixing Prolog and Forth

With the design described here, Forth and Prolog can be mixed freely because the Prolog machine is simulated directly in Forth. Prolog computations can be launched from Forth and Forth computations launched from Prolog. One way to mix the two would be to have the compiler recognize a distinguished functor (possibly **forth**) that would cause the Forth code enclosed in the following parentheses to be compiled in-line in the Prolog clause. For example, the definition of a Prolog procedure that takes a list L and, as a side effect, prints the time taken for a naive reverse of the list might look like:

```
test(L) :- forth(0 COUNT ! START.TIMER),
    nrev(L,L1),
    forth(STOP.TIMER COUNT @ CR ." number of microseconds ").
```

This would compile to the equivalent Forth:

```
1 VAR ENTER
0 COUNT ! START.TIMER
1 VAR 2 VAR 2 nrev CALL
STOP.TIMER COUNT @ CR ." number of microseconds " .
RETURN
```

With this approach, there is no overhead involved in mixed language programming; however, there is some ugliness in the interface. Another possibility is to provide a facility for the declaration of a Prolog interface to Forth. The syntax of such a declaration could be

forth_predicate(<Forth word>,<Prolog predicate>)

where the predicate has +'s and -'s in its argument positions to indicate input and output arguments respectively. For example, the declaration

forth_predicate('TEST',test(+,+,-))

would specify that a call to the Prolog procedure **test/3** would compile to code that would place the first two arguments on the Forth data stack, execute the Forth word `TEST`, bind/compare the top of the data stack with the third argument of the call and then either fail or succeed on the basis of the comparison. The cost of this approach is the overhead involved in transferring values between the Prolog control stack and the Forth data stack.

Conclusion

There are two major paths for extensions to the work reported here. The first leads to a very attractive delivery vehicle for real-time expert systems—Forth for the procedural component, Prolog for representation and reasoning. Forth's strengths in real-time applications are well known. Thus, the facility and efficiency with which abstract machines can be simulated in Forth makes the language an ideal platform on which to deliver real-time knowledge-based systems. A marriage of Prolog and Forth is currently being used for this purpose [PAL86]. Given a Forth engine, compiled Prolog on such a platform competes in performance with anything currently available and is likely to be superior in price-performance for a long time to come.

Extensions to the current work on the path leading to a fully formed delivery vehicle include additional PVM instructions of the sort mentioned above under optimizations, better indexing of clauses and more efficient use of control stack space. It would also be worthwhile to simulate the Warren Abstract Machine [WAR83] in Forth to understand the trade-offs between machine complexity and speed. Collaboration with Forth engine vendors could result in hardware features supporting high level languages built on a Forth platform. The Forth engine could even evolve into a Prolog engine.

The second path for extensions begins exploration of issues in computation only touched on by commercial Prolog implementations. The general thrust of the exploration is the extension of unification towards more sophisticated treatment of the objects to be unified. For example, unification in standard Prolog is based solely on the syntactic structure of terms. The language is untyped, and there is no notion of evaluation or co-reference (other than for logical variables). There are, however, numerous illustrations of how type systems (mapping terms into a user-supplied type lattice) can provide tremendous leverage in solving hard problems [WAL85]. Furthermore, absence of evaluation or co-reference in the unifier means that the terms 2 + 2 and 4 won't unify – not satisfactory behavior in an intelligent system.

There are thus proposals to extend Prolog in these and other directions ([KOR83], [SHA83]), and such extensions can be added on top of standard Prolog. Nevertheless, extending the language while keeping it efficient requires extending the underlying virtual machine. One of the interesting facets of the research by Warren, Clocksin and coworkers on Prolog compilation is the emphasis on reducing the problem of compiling Prolog to the problem of finding a concise clause description language. One research question, then, is how to elaborate the clause description language to handle the Prolog extensions in a natural way. The next question is how to build it.

This article is in the spirit of the earlier work on Prolog compilation, taking the position that compiled Prolog is an executable clause description and arguing, therefore, that Forth is a good choice for a PVM implementation language. Forth is an even better choice for compiler prototyping of the type required for exploratory Prolog extensions. Therefore, both of the research questions, developing descriptive locutions and simulating the underlying machine, can be tackled naturally in Forth.

Extension	Effort	Reference
Intelligent Backtracking	small	[KUM86]
Sorted Logic	small	[WAL85]
Logic with Equality	medium	[KOR83]
Parallel Logic Programming	large	[CON81]
Concept Unification	large	[KAH86]

Figure 16. Areas of Prolog Machine Extension.

Some specific areas of extension are indicated in Figure 16 along with estimates of the magnitude of the effort involved to extend the Forth code of Appendix A into an effective testbed. Intelligent backtracking may be one of the more straightforward extensions to implement [KUM86], and there is ample scope to develop and test new ideas in this area. Sorts and Kinds are clearly a powerful representation feature and might be implemented naturally by combining this Prolog with the object-oriented systems already available in Forth. Logic with equality would likely require more work than the former extensions, but the task is well bounded with significant theoretical issues to explore. Parallel logic programming requires significant effort, but there may be interesting applications to data acquisition and process control. Concept unification is the most ambitious extension – the general idea involves simulating the capability that people exhibit, for example, to unify the concepts of shoe and hammer in situations where the goal is to put a nail into the wall. Only by trying to make some of these extensions work will enough insight be gained to understand their value.

LISTING dans JEDI no 50

Sehr geehrte/r FORTH-Interessent/in,

für die verspätete Zusendung der Informationen und des Anmeldeformulares zur FORTH-Tagung bitte ich um Verständnis. Einige organisatorische Details konnten leider nicht früher geklärt werden.

Der Tagungsort ist ein Hörsaal des Karman-Auditoriums der RWTH-Aachen. Dieses Gebäude ist sehr zentral gelegen; man erreicht zu Fuß in fünf Minuten das Altstadtzentrum Aachens. Die gleiche Zeit benötigt man, um zum Lokal "Katakomben" in der Pontstraße 74-76 zu gelangen. Dort können Sie am Abend des Ankunftstages, am Samstagmittag, Samstagabend und am Sonntagmittag die Mahlzeiten gemeinsam mit anderen Teilnehmern einnehmen. An den Abenden stehen uns die Clubräume zur Verfügung, sodaß noch viel Zeit für Gespräche bleibt.

Eine gemeinsame Unterkunft wie in München konnte leider nicht gefunden werden. Schicken Sie daher umgehend die beiligende Karte mit Ihren Wünschen an den Aachener Verkehrsverein Bad Aachen e.V.; von dort erhalten Sie alle nötigen Informationen und die Reservierungsbestätigung zugeschickt. Studenten sollten auf der Karte den Wunsch nach besonders billigen Unterkünften vermerken, um Auskünfte über Jugendherbergen, Campingplätze o.ä. zu erhalten.

Informationen zur Anreise, zum Tagungsort und zum Verlauf der Tagung werden Ihnen bei Eingang der Anmeldung zugeschickt. Programme werden wir ab 15. März gesondert versenden.

WICHTIG: Vortragende sollten bis spätestens 15.2. ihre Anmeldung inkl. Abstract verschicken, damit wir den Tagungsablauf planen können. Falls die Aufnahme eines Artikels zum Vortrag in Proceedings gewünscht wird, muß dieser spätestens zum 31.3. druckfertig vorliegen. Die Proceedings werden zum Selbstkostenpreis bei der Registrierung erhältlich sein.

Wir freuen uns auf Ihr Kommen!

Bezüglich Nachfragen rufen Sie mich bitte (außer Do) zwischen 18.00Uhr und 20.00 Uhr an: R.Kretzschmar, 02401-4390

TITEL: DENOMINATION — Mr / Herr=0, Mme / Frau=1

NAME: NOM

VORNAME: PRENOM

FIRMA: STE

STRASSE: RUE

PLZ: CODE POSTAL ORT: COMMUNE

FG-MITGLIEDS-NR.:

Ich bezahle die folgenden Gebühren/Leistungen:
(Bitte Betrag einsetzen und addieren)

Teilnehmergebühr PARTICIPANT		DM 100,-	
Mitglied der FORTH-Gesellschaft MEMBRE FIG		DM 60,-	
Student (ohne festes Einkommen) ETUDIANT		DM 30,-	
Firmenvertreter SOCIETE		DM 120,-	
Vortragender	(Abzug)	DM 10,-	
Ausstellungsfläche STAND (m2)		DM/m² 100,-	
Vier Menüs 4 REPAS		DM 66,-	

Den berechneten Betrag... Summe TOTAL DM

☐ ...zahle ich durch beiliegenden Verrechnungsscheck. PAYÉ PAR CHEQUE CI-JOINT

☐ ...habe ich überwiesen auf das Girokonto / VIREMENT A: Kretzschmar Nr: 9 635 905 BLZ: 391 501 00 der Kreissparkasse Aachen/Zw.St. Baesweiler

☐ Ich möchte einen Vortrag halten. (30 Min.) INTERVENTION (30 min)

☐ Ich möchte ein Kurzreferat halten. (15 Min.) INTERVENTION (15 min)

☐ Ich möchte ein Poster ausstellen. (max. Größe: A1) AFFICHAGE (Taille maxi A1)

☐ Ich benötige ca. ☐ m² Ausstellfläche für: RESERVATION STAND EXPO POUR:

(weiter auf Rückseite)

Mein Vortrag soll mit folgendem Titel (plus Untertitel) in der Vortragsliste erscheinen: (Für eine Inhaltsübersicht bitte Rückseite benutzen!) SUJET DE L'INTERVENTION

FORTH-Tagungsbüro 89
c/o Rolf Kretzschmar
Rote Gasse 7

5112 Baesweiler

Anmeldung zur
FORTH-Tagung 1989
in Aachen, am 7., 8. und 9. April

Die Angaben im Anmeldevordruck werden in die Teilnehmerliste und Teilnehmerbescheinigung übernommen, die am Veranstaltungsort ausgehändigt werden.

Ort Datum Unterschrift

JEDI et FIG HAMBOURG (RFA)